# PARAMETERIZED FLOATING-POINT MODULES

Progress Report
31 October, 2001

*Pavle Belanović*
*Miriam Leeser*
*Northeastern University*
*Department of Electrical and Computer Engineering*

# Contents

# 1.0 Introduction

The aim of this project is to produce a library of fully pipelined and parameterized modules for performing floating-point arithmetic operations, targeted for implementation on Annapolis Microsystems reconfigurable computing engines. Following this, the library components are to be used to implement a solution to an application of interest using floating point arithmetic. Basic library components are Denormalize, Round and Normalize, Addition, Subtraction, Multiplication, Accumulation and Multiply Accumulate.

Section 2 of this report contains detailed progress information, as well as a timetable for completion of future goals. Following this is section 3, which briefly describes the IEEE floating point format in detail sufficient for the scope of this report. Also contained in this section is a discussion of our approach to parameterization of the floating point modules. All floating point modules are described in section 4 of this report. Accounts of functionality, pipelining and entity definitions for each module are included. Simulation and testing of each of the floating point modules are the subject of section 5. Section 6 contains a discussion on synthesis results to date. Future work is discussed in section 7 and the report is concluded in section 8.

## 2.0 Schedule and Progress

Status of each of the floating point modules in the library, for both simulation and synthesis, are given in Table 1.

| Module | Simulation | Synthesis | Tested in HW |
|---|---|---|---|
| Denormalize | ✓ | ✓ | ✓ |
| Round and Normalize | ✓ | ✓ | ✓ |
| Addition | ✓ | ✓ | ✓ |
| Subtraction | ✓ | ✓ | ✓ |
| Accumulation | Outstanding | Outstanding | Outstanding |
| Multiplication | Outstanding | Outstanding | Outstanding |
| Multiply Accumulate | Outstanding | Outstanding | Outstanding |

Table 1 : Status

Schedule for future work is shown in Table 2.

| Objective | Date |
|---|---|
| Synthesize Multiply module | 15 December 2001 |
| Simulate, synthesize and test Fix2Float and Float2Fix modules | 15 December 2001 |
| Parameterized K-Means using floating-point arithmetic | 15 December 2001 |

Table 2 : Future Work

# 3.0 Floating Point Formats

The IEEE single precision floating point number format specifies complete bitwidth of 32 bits, divided into (from MSB to LSB) 1 sign bit, 8 exponent bits and 23 mantissa bits. Denoting the sign bit with *s*, exponent with *e*, and mantissa with *f*, an IEEE single precision floating point number represents the real number:

$$(-1)^s \times (1.f) \times 2^{e-BIAS}$$

In the above equation, BIAS depends on the number of exponent bits and is 127 in the IEEE single precision floating point format. The 1 in front of the mantissa in the equation refers to the implied 1, which is always present when the number is normal. Since it is always present, it need not be represented explicitly and, to save bitwidth, it is not represented in the IEEE single precision floating point format.

To fully describe any floating point format, two quantities are needed: bitwidth of the exponent and that of the mantissa. The sign is always represented with 1 bit only. Hence, to fully parameterize any floating point operation, two parameters are sufficient. Within this project, the two parameters will be referred to as `exp_bits` for exponent bitwidth and `man_bits` for mantissa bitwidth. Implementation of these parameters in the VHDL descriptions of the parameterized modules is achieved through `generic` statements.

The two parameters are fixed at synthesis time by the top level module which instantiates them with appropriate values. Therefore, these parameters are not inputs to, or outputs from, any of the modules and are thus not represented in their entity diagrams.

# 4.0 Floating Point Modules

Modules shown in Table 2 form the floating point library. Also indicated in the table are the number of pipeline stages in the module, or the latency of the module in clock cycles.

| Module | Pipeline Stages |
|---|---|
| Denormalize | 0 |
| Round and normalize | 2 |
| Addition | 4 |
| Subtraction | 4 |
| Accumulation | ? |
| Multiplication | ? |
| Multiply Accumulate | ? |

Table 3 : Floating Point Modules

The above floating point modules are fully pipelined. To facilitate their connection into longer pipelines, each module is provided with a READY input, indicating data on the inputs to the module is valid. Also, each module has a DONE output, which is active high when output from the module is valid.

Exceptions may arise in floating point calculations and may affect part of, or the entire pipeline. Thus, each module is capable of handling two exception conditions:

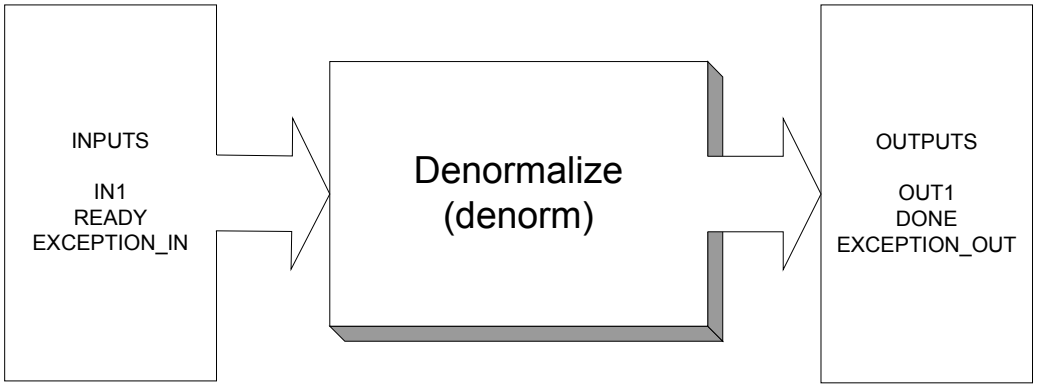
**External source of exception** – When an exception occurs earlier on in the pipeline, upstream from the module, it is important for that exception condition to percolate down to the end of the pipeline. Thus, the module will accept the exception at its input, not process incoming data as it is not valid, and pass on the exception condition to the rest of the pipeline.

**Internal source of exception** – Exceptions may arise during the operation of the floating point module. In such instances, the module will generate an exception condition and pass it on to the rest of the pipeline.

To implement both of these exception conditions, each module has an EXCEPTION_IN input and an EXCEPTION_OUT output.

## 4.1 Denormalize (denorm)

The function of the denormalize module is to explicitly introduce the implied ‘1’ in the mantissa, as represented in “ $1.f$ ” in the equation above.
The entity of the denormalize module (denorm) is shown below.




The denormalize module is meant to be placed at the beginning of a pipeline of modules, so as to "unpack" a floating point number by introducing the implied '1'. This is meant to be reversed by the round and normalize module, as explained below. The denormalize module, due to its simplicity, is a purely combinational circuit and does not take a clock cycle to perform.

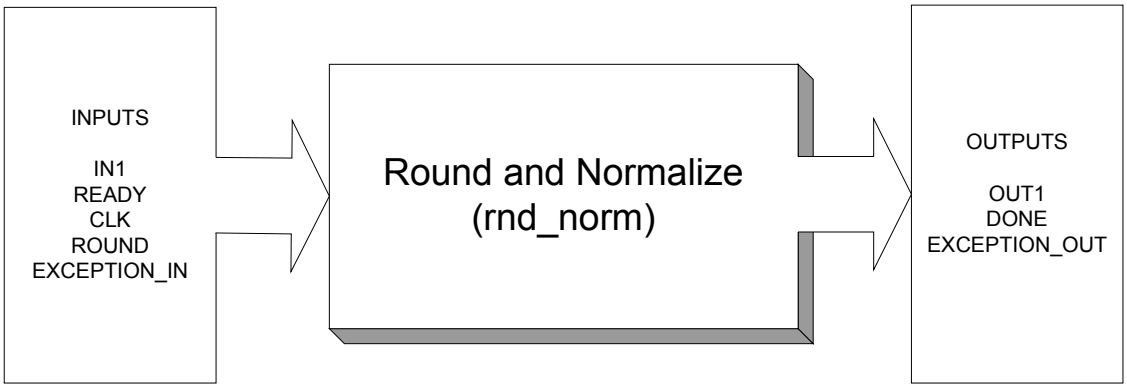
## 4.2 Round and Normalize (rnd_norm)

For a pipeline of modules to produce a normalized floating point number on the output, it must have a round and normalize module at its end. This module performs several functions:

- Normalizing the input
- Performing a rounding addition, if necessary
- Truncating the output to the appropriate bitwidth

Implementation of the round and normalize module is structural, composed of two sub-modules: normalizer (to perform the first function in the list above) and rounding addition (for the latter two functions). The 'round' input is used to select the rounding mode, as explained in section 4.2.2.

The entity of the round and normalize module is shown below.

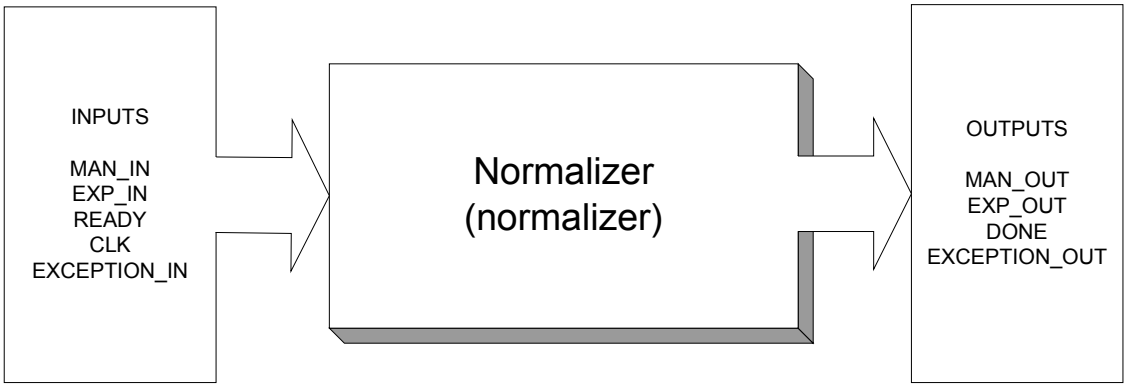



### 4.2.1 Normalizer (normalizer)

The first function of the round and normalize module is to normalize the input, which involves two operations: shifting the mantissa to the left, until its MSB is '1' and subtracting from the exponent the number of times the mantissa was shifted left. For example:

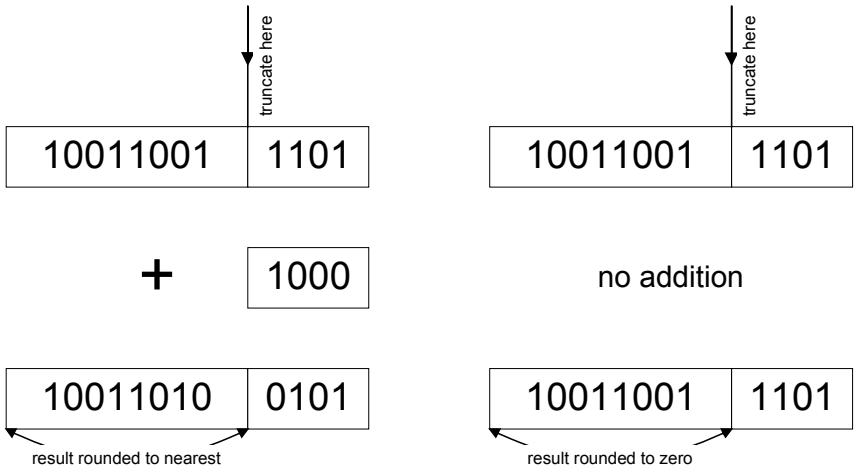
$$0.012887 \times 10^{8} = 1.2887 \times 10^{6}$$

In the above example, the mantissa is shifted left twice, and thus the exponent has 2 subtracted from it. The entity of the normalizer module is shown below.

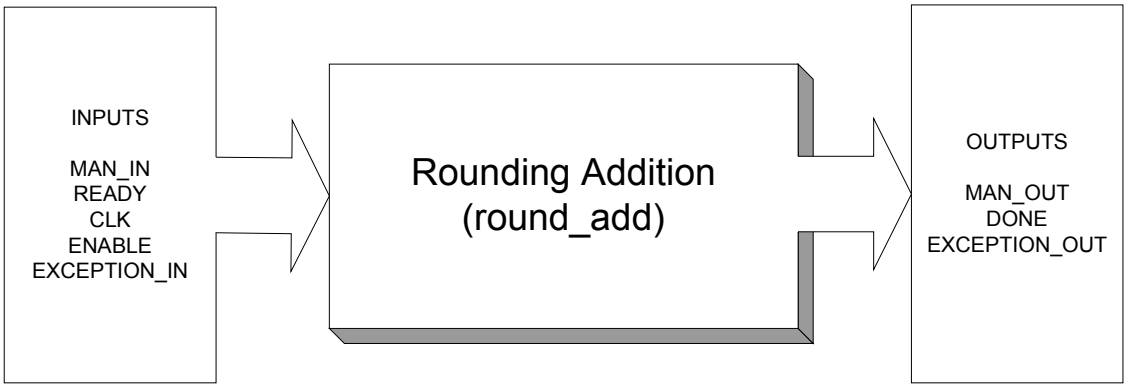



### 4.2.2 Rounding Addition (round_add)

Implemented rounding schemes in this project are round to nearest (default) and round to zero. Rounding, of course, implies throwing away some bits of the mantissa and the rounding schemes deal with this truncation in different ways. To round to nearest, we add a '1' to the MSB of the slice of the mantissa to be thrown away, and then truncate. To round to zero, we just truncate without the addition.




To choose between the rounding schemes, a 1-bit ('round') input into the module is sufficient. The input is encoded as: '0' signals round to zero, '1' signals round to nearest. Thus, this input is used as an active high enable signal into an adder that adds the '1' to the MSB of the slice of mantissa to be thrown away. The rouding addition module also truncates the output to the required bitwidth. Given below is the entity for the rounding addition module.

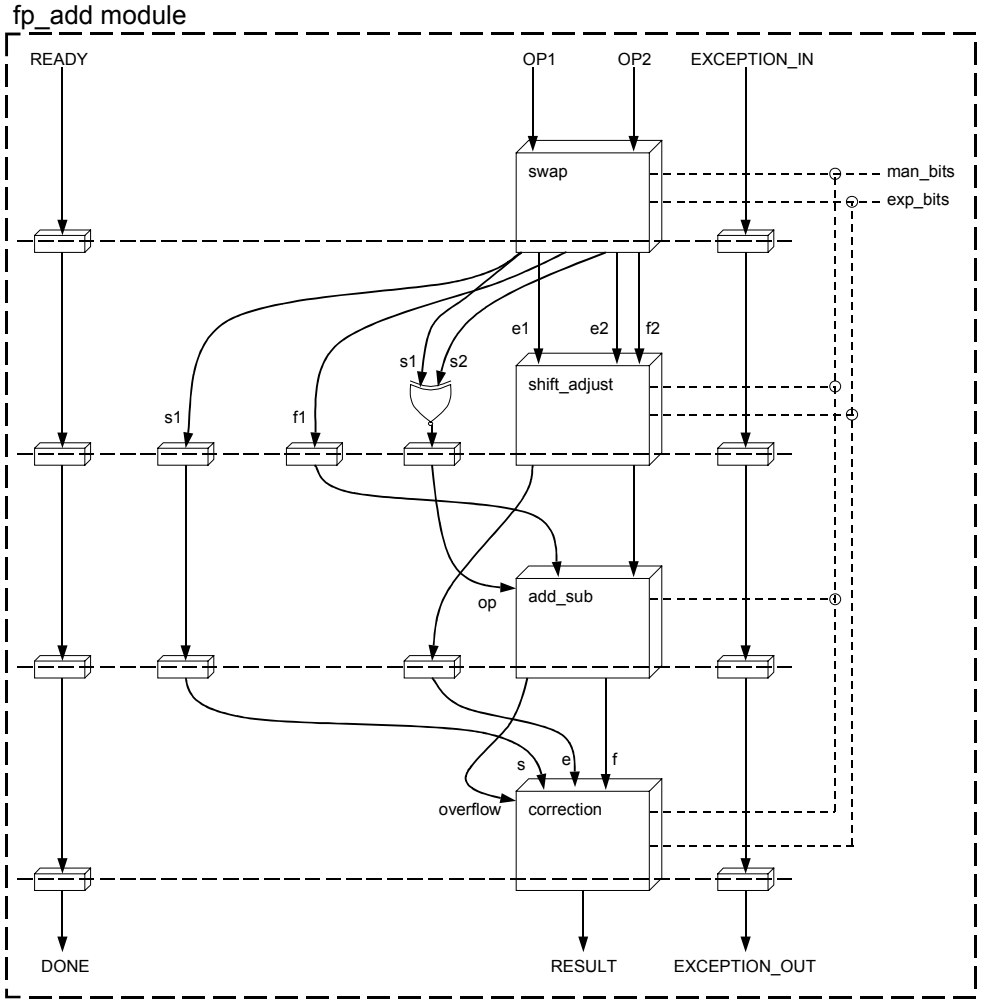


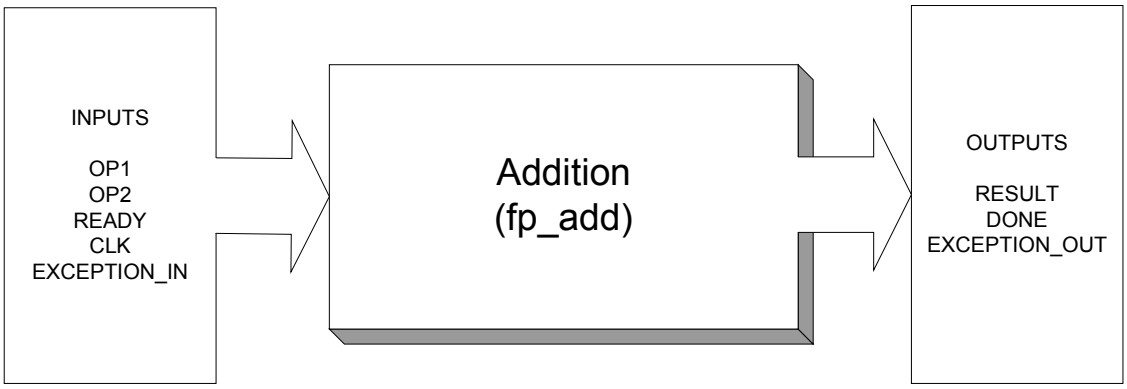

## 4.3 Addition (fp_add)

Floating point addition algorithm is composed of the following steps:

- Make sure the input with larger magnitude is on input 1 (swap)
- Align the mantissas (shift and adjust)
- Add or subtract the mantissas (add/subtract)
- Shift mantissa right and increment the exponent if overflow in addition occurred (correction)

The addition module (fp_add) is described structurally from the above modules, as shown below.

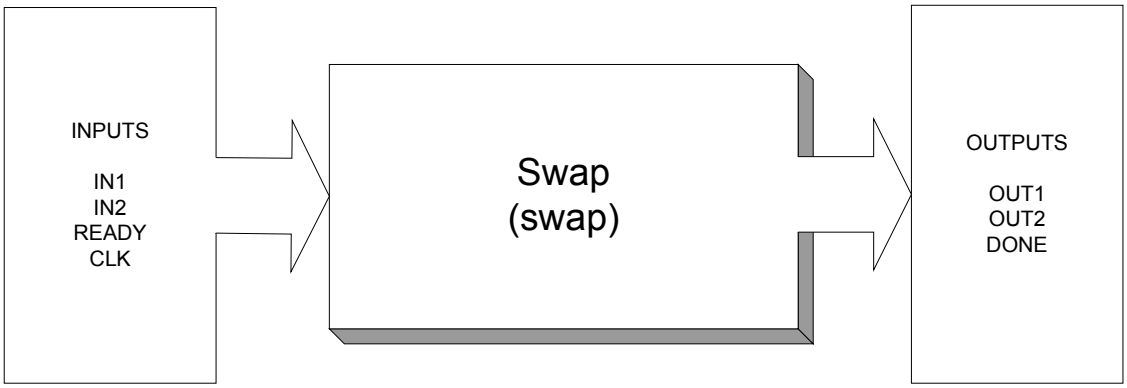





Please note that the addition operation is performed with one guard bit. Hence, the bitwidth of the mantissa, and thus the overall result, at the output of the addition module is one bit wider than at its input.

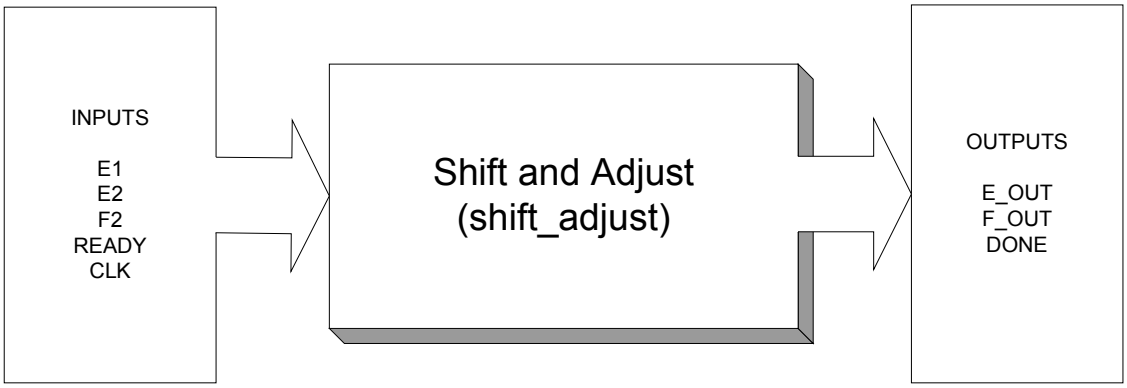
### 4.3.1 Swap (swap)

Processing within the addition algorithm needs to assume that one of the inputs, say input 1, always has magnitude higher or equal to that of the other input, in this case input 2. Since the inputs have to be sorted into a large one and a small one, they may need to be swapped. This conditional swap is the function of the swap module. Its entity is given below.




### 4.3.2 Shift and Adjust (shift_adjust)

Mantissas of the two inputs are to be added, but they have to be appropriately aligned before the addition. In other words, the mantissa of the smaller number (f2) has to be aligned to match the mantissa of the larger number (f1). This translates into shifting the smaller number's mantissa (f2) right. The difference between the exponents, (e1-e2), is how many spaces the mantissa f2 is to be shifted. This number is always non-negative, since the swap module ensures $e1 \geq e2$. The shift and adjust module performs this variable shift function. Its entity is shown below.

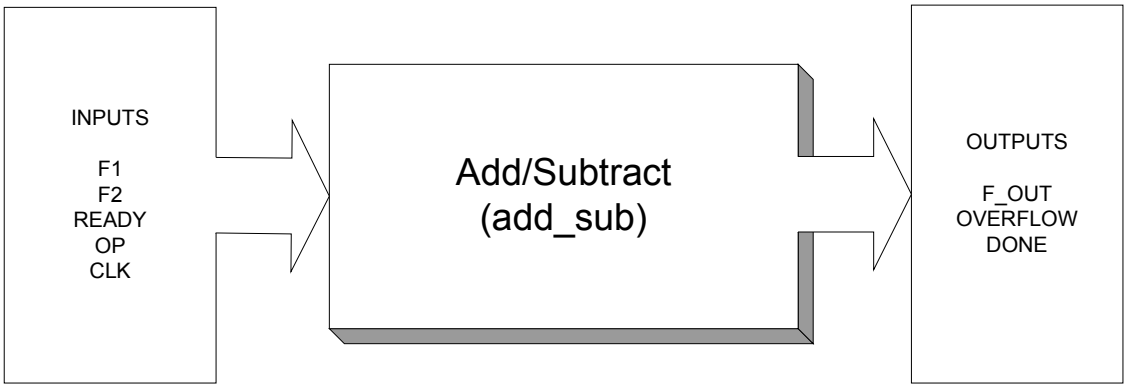



### 4.3.3 Add/Subtract (add_sub)

The aligned mantissas of the two inputs have to be added or subtracted to produce the mantissa of the final result. The exponent of the final result is the exponent of the larger input, since both mantissas were aligned to this exponent. The operation to be performed depends on the signs of the two inputs. The operation is encoded with one bit, op, as follows.

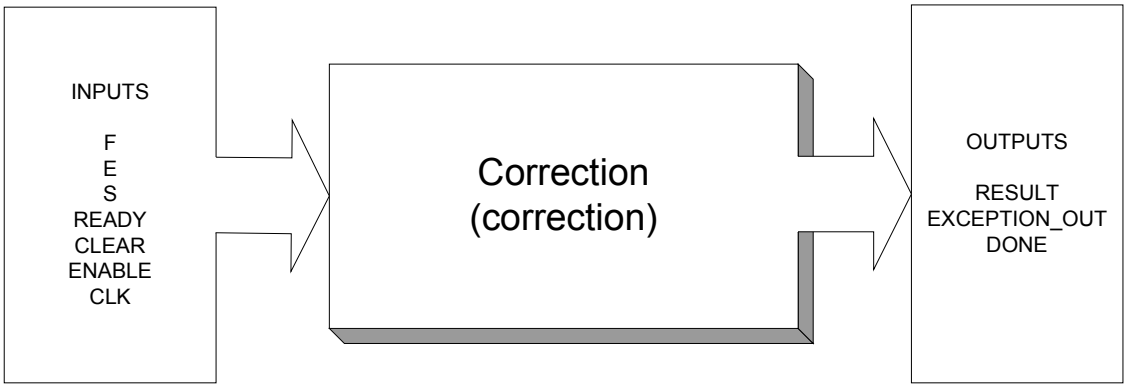
| s1 | | s2 | | op | |
|---|---|---|---|---|---|
| + | 0 | + | 0 | + | 1 |
| + | 0 | - | 1 | - | 0 |
| - | 1 | + | 0 | - | 0 |
| - | 1 | - | 1 | + | 1 |

The above table indicates op = s1 XNOR s2 . The add/subtract module also passes on the information to the module correction on whether an overflow occurred during addition. The entity of the add_sub module is given below.




### 4.3.4 Correction (correction)

If an overflow occurs during addition of mantissas, it is necessary to correct the result. This operations involves two steps: right shift result mantissa by one place, filling the MSB with '1', and incrementing the result exponent to reflect the shift. Hence, the overflow signal of the add/subtract module acts as an enable of the correction module which will either correct the incoming exponent and mantissa or just pass them through as they are. In its current version, this module incorporates checking for overflow on incrementing the exponent. The entity of the correction module is given below.




## 4.4 Subtraction (fp_sub)

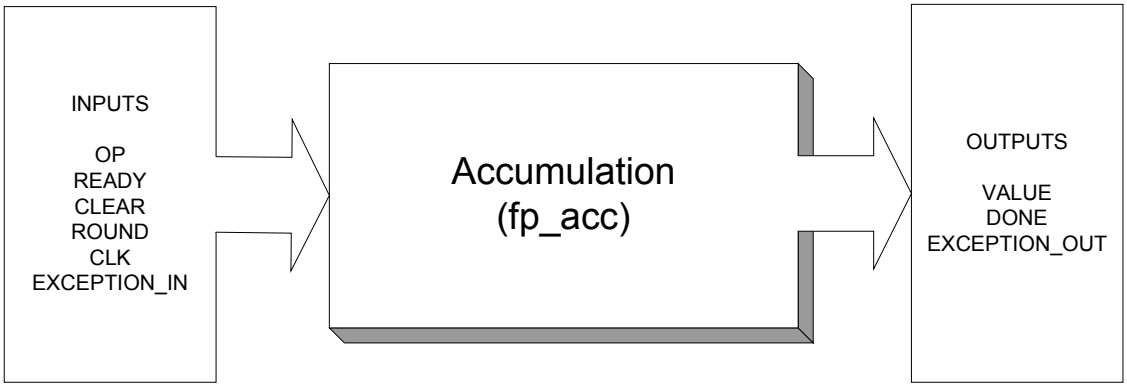
Because of the similarity of the addition and subtraction operations, especially in the floating point format, it is possible to reuse the entire structure of the fp_add module to construct the fp_sub module. Because:

$$a - b = a + (-b),$$

the only structural difference between the two modules is a single inverter on the sign bit of the second input in the subtraction module (fp_sub).

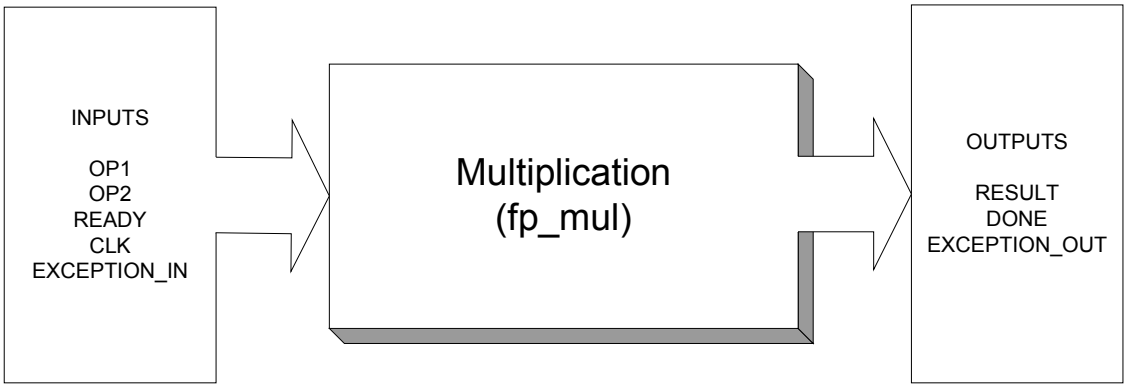
## 4.5 Accumulation (fp_acc)

Accumulation of floating-point numbers is performed by the fp_acc module. This module has yet to be implemented in either simulation or hardware at this time. The entity of the accumulator module is shown below.



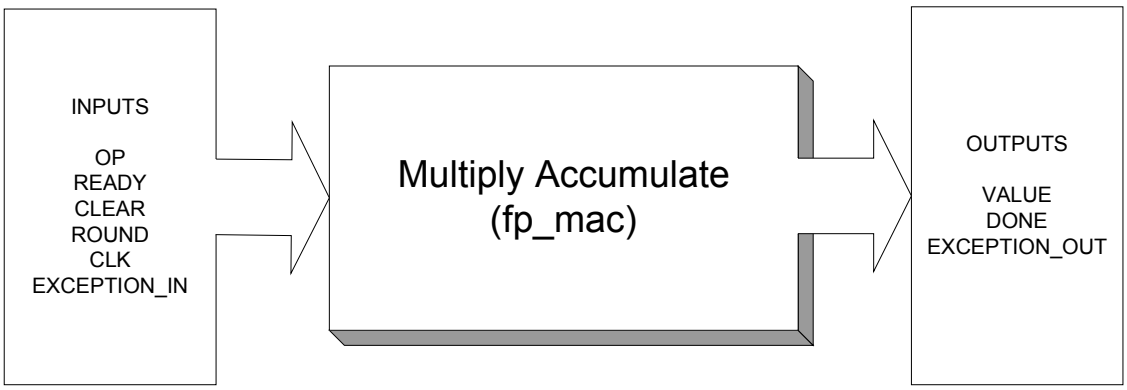

## 4.6 Multiplication (fp_mul)

The multiplication module computes the product of two floating point numbers. This module is composed of three pipeline stages. Currently, simulation and testing of this component are complete; synthesis of the module is scheduled for near future (see section 2). Entity of the fp_mul module is as shown below.




## 4.7 Multiply Accumulate (fp_mac)

The multiply accumulate module has yet to be implemented in either simulator or hardware at this time. This module will be analogous to the accumulate module and will have an entity as represented below.

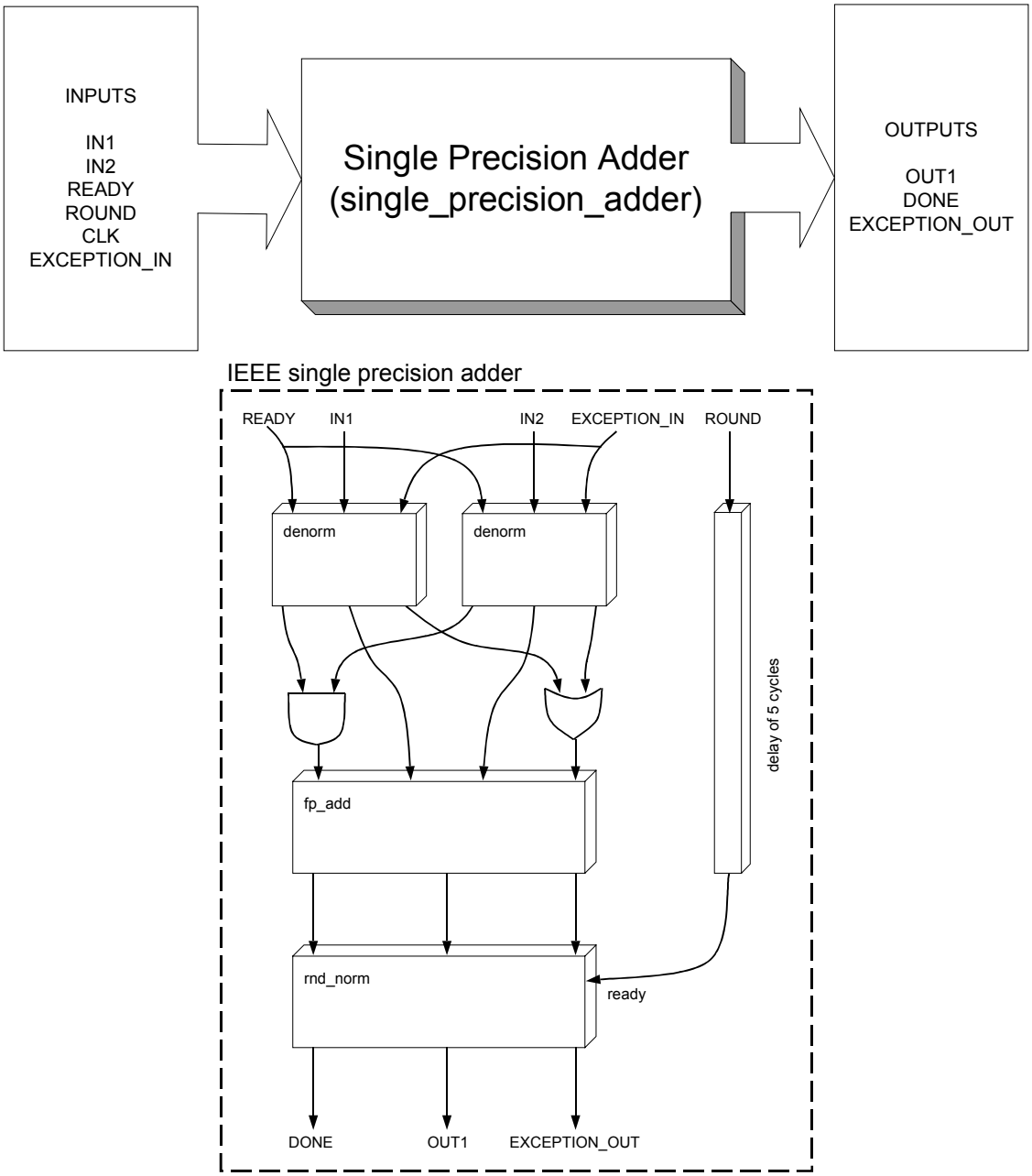



## 4.8 Other Modules

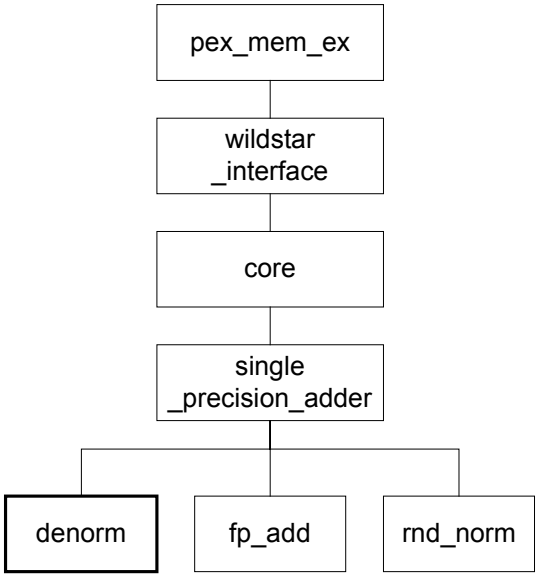
Other modules in the project, not described above, are non-parameterized instances in the design space, such as single_precision_adder or half_precision_subtractor. These are top-level modules, encompassing several of the parameterized modules described above. All these modules are constructed as examples of how to pipeline the parameterized floating point modules into working, implementation-ready modules.

### 4.8.1 Single Precision Adder (single_precision_adder)

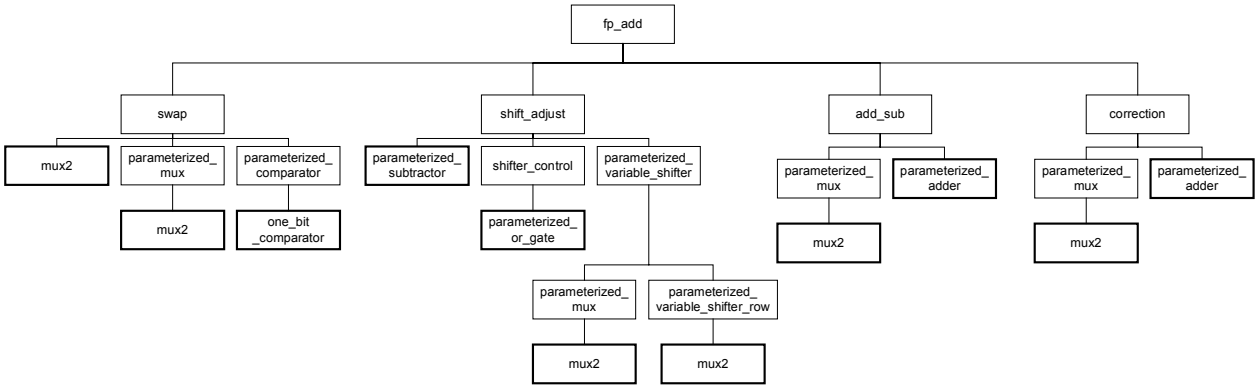
The single_precision_adder module is described in VHDL structurally, instantiating two denormalize modules, one addition module and one round and normalize module. All these modules are instantiated to mantissa bitwidth of 23 bits and exponent bitwidth of 8 bits. This is the IEEE single precision format. The structure and the entity of the single_precision_adder are shown below.



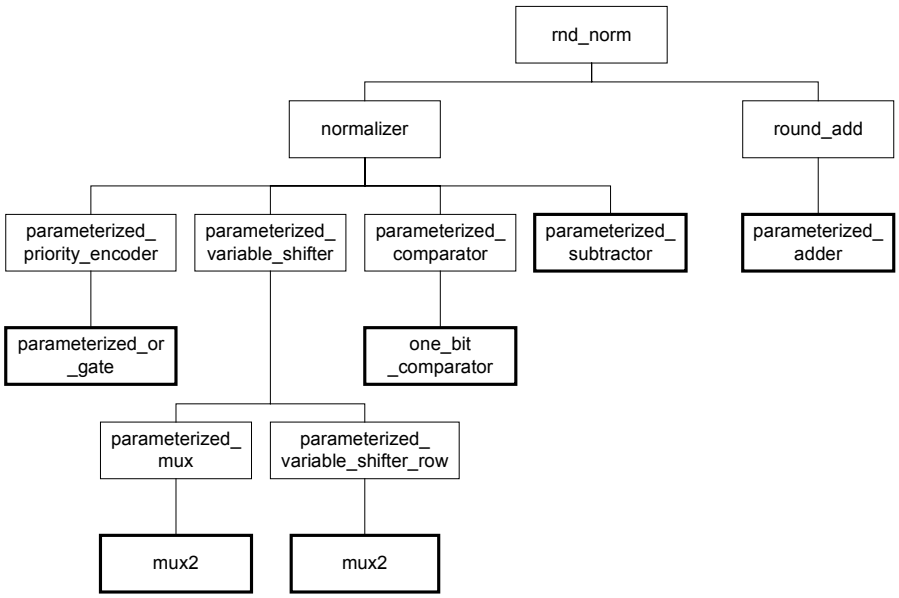

Since single_precision_adder is a top level module, it is found at the top of the hierarchy tree, as shown below. Please note that the modules with a heavy box outline, found at the leaves of the tree, are primitives, described at gate level.




Hierarchy tree for implementation of the single_precision_adder module




Hierarchy tree of fp_add sub-module




Hierarchy tree of rnd_norm sub-module

# 5.0 Simulation and Testing

The design cycle for each parameterized floating point module calls first for a simulation and testing stage, followed by a synthesis stage. This section is dedicated to the simulation and testing stage of each of the modules.

After each module has been described in VHDL, it was necessary to debug and verify the description using a simulator, which is ModelSim/Plus 5.2e for this project. To ensure completeness of debugging at this stage, and thus minimize time spent debugging at the synthesis stage, a set of test vectors was developed for each of the floating point modules and sub-modules in this project, all the way down the hierarchy and including the primitives (such as full_adder for example).

The remainder of this section lists all the parameterized floating point modules that have been taken through the simulation and testing stage of their design. This will include all the first level sub-modules. For each module and sub-module, we include test cases and corresponding test vectors.

## 5.1 Denormalize (denorm)

The functionality of the denormalize module is relatively simple; it has to widen the datapath by one bit. It does so by inserting a logic high value, '1', between the exponent and mantissa sections of the floating point number. In other words, the denormalize module makes explicit the implied '1' of the mantissa of a normal floating point number. Hence, there is only one case to test for: insertion of the implied '1'.

IN1: $150FBD15_{16}$ = 0001 0101 0000 1111 1011 1101 0001 $0101_2$
OUT1: $02A8FBD15_{16}$ =0 0010 1010 1000 1111 1011 1101 0001 $0101_2$

The denormalize module generates an exception if the input has an all-zero exponent.

## 5.2 Round and Normalize (rnd_norm)

Round and normalize module (rnd_norm) consists of two sub-modules: normalizer and rounding addition. In functionality, the round and normalize module takes in a floating point number, which is not guaranteed normal, and outputs a normal, narrower, rounded number, which does not include the implied '1'. The input bitwidth is always wider than the output bitwidth, and the narrowing of the datapath is achieved through the rounding method specified by the round input.

Hence, the test cases for this module are:

- no normalization, round to zero,
- normalization, round to zero,
- no normalization, round to nearest,
- normalization, round to nearest.

Test vectors and results for each case are given below respectively, for a scenario of 1-7-12 (s-e-f) 20-bit floating point number being input, narrowed to a 1-7-8 16-bit floating point number at the output.

IN1: $5A8FB_{16}$ = 0101 1010 1000 1111 $1011_2$
ROUND: 0
OUT: $5A1F_{16}$ = 0101 1010 0001 $1111_2$

IN1: $DB47D_{16}$ = $1101\ 1011\ 0100\ 0111\ 1101_2$
ROUND: 0
OUT: $DA1F_{16}$ = $1101\ 1010\ 0001\ 1111_2$

IN1: $5A8FB_{16}$ = $0101\ 1010\ 1000\ 1111\ 1011_2$
ROUND: 1
OUT: $5A1F_{16}$ = $0101\ 1010\ 0001\ 1111_2$

IN1: $DB47D_{16}$ = $1101\ 1011\ 0100\ 0111\ 1101_2$
ROUND: 1
OUT: $DA20_{16}$ = $0101\ 1010\ 0010\ 0000_2$

### 5.2.1 Normalizer (normalizer)

The normalizer module has the function of normalizing the floating point number at its input. This operation involves two parts: shifting the mantissa left until its MSB is '1', and decrementing the exponent to reflect the shift in the mantissa. Please note that if the input number is already normal, the output is equal to the input. Hence, the test cases for this module are:

- no normalization required,
- normalize.

Below given are test vectors for these two cases respectively.

MAN_IN: $8C_{16}$ = $1000\ 1100_2$
EXP_IN: $A_{16}$ = $1010_2$
MAN_OUT: $8C_{16}$ = $1000\ 1100_2$
EXP_OUT: $A_{16}$ = $1010_2$

MAN_IN: $23_{16}$ = $0010\ 0011_2$
EXP_IN: $A_{16}$ = $1010_2$
MAN_OUT: $8C_{16}$ = $1000\ 1100_2$
EXP_OUT: $8_{16}$ = $1000_2$

### 5.2.2 Rounding Addition (round_add)

The rounding addition stage of the round normalize module has the function of enabled addition, as explained in section 4.2.2. The input to the module is a normal number, while the output is a normal and rounded number with the MSB of the mantissa, which is assumed '1', removed. Test cases for this module are:

- round to zero (no addition),
- round to nearest, no effect,
- round to nearest, with effect.

Given below are test vectors using a scenario of 12-bit mantissa being narrowed to an 8-bit mantissa.

MAN_IN: $9DB_{16}$ = $1001\ 1101\ 1011_2$
ENABLE: 0

MAN_OUT: $3B_{16}$ = 0011 $1011_2$

MAN_IN: $9DB_{16}$ = 1001 1101 $1011_2$
ENABLE: 1
MAN_OUT: $3B_{16}$ = 0011 $1011_2$

MAN_IN: $9DF_{16}$ = 1001 1101 $1111_2$
ENABLE: 1
MAN_OUT: $3C_{16}$ = 0011 $1100_2$

## 5.3 Addition (fp_add)

The addition module, fp_add, is composed of 4 sub-modules: swap, shift_adjust, add_sub and correction. Given two denormalized floating-point numbers, the fp_add module will produce their sum as a 1 bit wider floating point number. Test cases for this module are:

- addition, no swap, same exponent, no overflow,
- addition, no swap, same exponent, overflow,
- addition, no swap, small difference in exponents,
- addition, no swap, large difference in exponents,
- addition, swap, small difference in exponents,
- subtraction, no swap, small difference in exponents,
- subtraction, swap, small difference in exponents,
- subtraction, no swap, large difference in exponents.

OP1: $5D4BDA51_{16}$ = 0101 1101 0100 1011 1101 1010 0101 $0001_2$
OP2: $5D0ADBF2_{16}$ = 0101 1101 0000 1010 1101 1011 1111 $0010_2$
RESULT: $0BAAD6C86_{16}$ = 0 1011 1010 1010 1101 0110 1100 1000 $0110_2$

OP1: $5D4BDA51_{16}$ = 0101 1101 0100 1011 1101 1010 0101 $0001_2$
OP2: $5D4ADBF2_{16}$ = 0101 1101 0100 1010 1101 1011 1111 $0010_2$
RESULT: $0BB96B643_{16}$ = 0 1011 1011 1001 0110 1011 0110 0100 $0011_2$

OP1: $5D4BDA51_{16}$ = 0101 1101 0100 1011 1101 1010 0101 $0001_2$
OP2: $5C4ADBF2_{16}$ = 0101 1100 0100 1010 1101 1011 1111 $0010_2$
RESULT: $0BABD229B_{16}$ = 0 1011 1010 1011 1101 0010 0010 1001 $1011_2$

OP1: $5D4BDA51_{16}$ = 0101 1101 0100 1011 1101 1010 0101 $0001_2$
OP2: $4C4ADBF2_{16}$ = 0100 1100 0100 1010 1101 1011 1111 $0010_2$
RESULT: $0BA97B4A2_{16}$ = 0 1011 1010 1001 0111 1011 0100 1010 $0010_2$

OP1: $5D4BDA51_{16}$ = 0101 1101 0100 1011 1101 1010 0101 $0001_2$
OP2: $5DCADBF2_{16}$ = 0101 1101 1100 1010 1101 1011 1111 $0010_2$
RESULT: $0BBE19235_{16}$ = 0 1011 1011 1110 0001 1001 0010 0011 $0101_2$

OP1: $5D4BDA51_{16}$ = 0101 1101 0100 1011 1101 1010 0101 $0001_2$
OP2: $DCC3DA51_{16}$ = 1101 1100 1100 0011 1101 1010 0101 $0001_2$
RESULT: $0BA53DA51_{16}$ = 0 1011 1010 0101 0011 1101 1010 0101 $0001_2$

OP1: $5D4BDA51_{16}$ = 0101 1101 0100 1011 1101 1010 0101 $0001_2$
OP2: $DDC3DA51_{16}$ = 1101 1101 1100 0011 1101 1010 0101 $0001_2$

RESULT: $1BB3BDA51_{16}$ = 1 1011 1011 0011 1011 1101 1010 0101 $0001_2$

OP1: $5D4BDA51_{16}$ = 0101 1101 0100 1011 1101 1010 0101 $0001_2$
OP2: $C5C3DA51_{16}$ = 1100 0101 1100 0011 1101 1010 0101 $0001_2$
RESULT: $0BA97B4A2_{16}$ = 0 1011 1010 1001 0111 1011 0100 1010 $0010_2$

### 5.3.1 Swap (swap)

The first part of the addition algorithm for floating point numbers demands that the operands be sorted into the larger and the smaller one, based on magnitude. This operation is a conditional swap. Since magnitude depends on the exponent primarily and the mantissa secondarily, five cases need to be tested:

- different exponents, no swap, (based on exponent)
- different exponents, swap, (based on exponent)
- same exponent, no swap, (based on exponent, mantissa)
- same exponent, swap (based on exponent, mantissa)
- same number.

Test vectors for these cases are given below.

IN1: $58E9AE29_{16}$ = 0101 1000 1110 1001 1010 1110 0010 $1001_2$
IN2: $D77B0A41_{16}$ = 1101 0111 0111 1011 0000 1010 0100 $0001_2$
OUT1: $58E9AE29_{16}$ = 0101 1000 1110 1001 1010 1110 0010 $1001_2$
OUT2: $D77B0A41_{16}$ = 1101 0111 0111 1011 0000 1010 0100 $0001_2$

IN1: $D77B0A41_{16}$ = 1101 0111 0111 1011 0000 1010 0100 $0001_2$
IN2: $58E9AE29_{16}$ = 0101 1000 1110 1001 1010 1110 0010 $1001_2$
OUT1: $58E9AE29_{16}$ = 0101 1000 1110 1001 1010 1110 0010 $1001_2$
OUT2: $D77B0A41_{16}$ = 1101 0111 0111 1011 0000 1010 0100 $0001_2$

IN1: $D77B0A41_{16}$ = 1101 0111 0111 1011 0000 1010 0100 $0001_2$
IN2: $5769AE29_{16}$ = 0101 0111 0110 1001 1010 1110 0010 $1001_2$
OUT1: $D77B0A41_{16}$ = 1101 0111 0111 1011 0000 1010 0100 $0001_2$
OUT2: $5769AE29_{16}$ = 0101 0111 0110 1001 1010 1110 0010 $1001_2$

IN1: $5769AE29_{16}$ = 0101 0111 0110 1001 1010 1110 0010 $1001_2$
IN2: $D77B0A41_{16}$ = 1101 0111 0111 1011 0000 1010 0100 $0001_2$
OUT1: $D77B0A41_{16}$ = 1101 0111 0111 1011 0000 1010 0100 $0001_2$
OUT2: $5769AE29_{16}$ = 0101 0111 0110 1001 1010 1110 0010 $1001_2$

IN1: $5769AE29_{16}$ = 0101 0111 0110 1001 1010 1110 0010 $1001_2$
IN2: $5769AE29_{16}$ = 0101 0111 0110 1001 1010 1110 0010 $1001_2$
OUT1: $5769AE29_{16}$ = 0101 0111 0110 1001 1010 1110 0010 $1001_2$
OUT2: $5769AE29_{16}$ = 0101 0111 0110 1001 1010 1110 0010 $1001_2$

### 5.3.2 Shift and Adjust (shift_adjust)

The shift and adjust module aligns the mantissas of the two operands by shifting the smaller operand's mantissa right. The amount of shift required is determined through a subtraction of the two exponents. If the difference in the exponents is larger than the number of mantissa bits, the smaller mantissa is shifted

out of bounds, i.e. becomes all zeros. Bitwidth of the mantissa expands by one (guard) bit. Thus, the test cases for this module are:

- no shift required (equal exponents),
- shift within bounds (small difference in exponents),
- shift right on bound (critical difference in exponents),
- shift out of bounds (large difference in exponents).

Below are test vectors and results for above test cases respectively.

E1: $A_{16}$ = $1010_2$
E2: $A_{16}$ = $1010_2$
F2: $8B_{16}$ = $1000\ 1011_2$
F_OUT: $116_{16}$ = $1\ 0001\ 0110_2$

E1: $8_{16}$ = $1000_2$
E2: $3_{16}$ = $0011_2$
F2: $CA_{16}$ = $1100\ 1010_2$
F_OUT: $00C_{16}$ = $0\ 0000\ 1100_2$

E1: $A_{16}$ = $1010_2$
E2: $1_{16}$ = $0001_2$
F2: $AB_{16}$ = $1010\ 1011_2$
F_OUT: $000_{16}$ = $0\ 0000\ 0000_2$

E1: $F_{16}$ = $1111_2$
E2: $4_{16}$ = $0100_2$
F2: $BA_{16}$ = $1011\ 1010_2$
F_OUT: $000_{16}$ = $0\ 0000\ 0000_2$

### 5.3.3 Add/Subtract (add_sub)

The add/subtract module is used in adding or subtracting mantissas of the two operands. An addition operation may or may not produce an overflow (‘carry out’ of the adder). Thus, the test cases for the add/subtract module are:

- addition to zero,
- addition, no overflow,
- addition, overflow,
- subtraction.

F1: $AB_{16}$ = $1010\ 1011_2$
F2: $00_{16}$ = $0000\ 0000_2$
OP: 1
F_OUT: $AB_{16}$ = $1010\ 1011_2$
OVERFLOW: 0

F1: $AB_{16}$ = $1010\ 1011_2$
F2: $10_{16}$ = $0001\ 0000_2$
OP: 1
F_OUT: $BB_{16}$ = $1011\ 1011_2$

OVERFLOW: 0

F1: $AB_{16}$ = $1010\ 1011_2$
F2: $80_{16}$ = $1000\ 0000_2$
OP: 1
F_OUT: $2B_{16}$ = $0010\ 1011_2$
OVERFLOW: 1

F1: $AB_{16}$ = $1010\ 1011_2$
F2: $80_{16}$ = $1000\ 0000_2$
OP: 0
F_OUT: $2B_{16}$ = $0010\ 1011_2$
OVERFLOW: 0

### 5.3.4 Correction (correction)

The operation of the correction module is that of shifting the mantissa of the result exactly one bit to the right, filling the vacant MSB with '1'. Also, the exponent must be incremented to reflect this shift. This operation is enabled by the overflow output of the add/subtract module. Also, in the case of an exception, the output of the correction module is set to all zeros. Hence, the test cases for this module are:

- no correction,
- correction,
- clear.

Listed below are test vectors and results for the cases above.

S: 1
E: $A_{16}$ = $1010_2$
F: $5F_{16}$ = $0101\ 1111_2$
ENABLE: 0
CLEAR: 0
RESULT: $1A5F_{16}$ = $1\ 1010\ 0101\ 1111_2$

S: 0
E: $A_{16}$ = $1010_2$
F: $5F_{16}$ = $0101\ 1111_2$
ENABLE: 1
CLEAR: 0
RESULT: $0BAF_{16}$ = $0\ 1011\ 1010\ 1111_2$

S: 0
E: $A_{16}$ = $1010_2$
F: $5F_{16}$ = $0101\ 1111_2$
ENABLE: 1
CLEAR: 1
RESULT: $0000_{16}$ = $0\ 0000\ 0000\ 0000_2$

## 5.4 Subtraction (fp_sub)

The operation of the fp_sub module is virtually identical to the operation of the fp_add module, since their structure is the same, except the fp_sub module includes an inverter on the sign bit of the second operand. Hence, the operation of this module, as well as all its sub-modules has already been verified. Given below is an example of IEEE single precision subtraction, as executed by the module single_precision_subtractor, which includes the the fp_sub module.

| Decimal | | IEEE single precision number |
|---|---|---|
| 23.279462 | => | 41BA3C57 |
| 201.779142 | => | 4349C776 |

23.279462 – 201.779142 = -178.499680 (real answer)

Answer from module = C3327FEC = -178.499694824 Correct to 5th decimal place.

## 5.5 Multiplication (fp_mul)

Module fp_mul has been designed and tested in simulation, but has not been implemented in hardware. It performs a floating point multiplication of its two operands. Thus, the test cases for this module are:

- multiplication,
- exception in.

Given below are test vectors for the two test cases, based on the scenario of 1-6-9 (s-e-f) format floating point numbers as inputs.

| | | |
|---|---|---|
| OP1: | $4772_{16}$ = | $0100\ 0111\ 0111\ 0010_2$ |
| OP2: | $5176_{16}$ = | $0101\ 0001\ 0111\ 0110_2$ |
| EXC.IN: | 0 | |
| RESULT: | $5B0E_{16}$ = | $0101\ 1011\ 0000\ 1110_2$ |

| | | |
|---|---|---|
| OP1: | $4772_{16}$ = | $0100\ 0111\ 0111\ 0010_2$ |
| OP2: | $5176_{16}$ = | $0101\ 0001\ 0111\ 0110_2$ |
| EXC.IN: | 0 | |
| RESULT: | $5B0E_{16}$ = | $0101\ 1011\ 0000\ 1110_2$ |

## 5.6 Other Modules

The only high-level, non-parameterized module simulated and tested in this project is the single_precision_adder.

### 5.6.1 Single Precision Adder (single_precision_adder)

This module incorporates instances of the denorm, fp_add and rnd_norm modules. All of these parameterized modules have been tested, as shown above. Test cases for the non-parameterized single_precision_adder module are:

- both operands positive,
- one positive, other negative,

- both negative,
- one much larger than other,
- same exponent, causing correction module to work.

Below is a set of test vectors and results for each of the above cases, respectively.

OP1: $4B2D064F_{16}$ = $0100\ 1011\ 0010\ 1101\ 0000\ 0110\ 0100\ 1111_2$
OP2: $4A479127_{16}$ = $0100\ 1010\ 0100\ 0111\ 1001\ 0001\ 0010\ 0111_2$
RESULT: $4B5EEA99_{16}$ = $0100\ 1011\ 0101\ 1110\ 1110\ 1010\ 1001\ 1001_2$

OP1: $CB2D064F_{16}$ = $1100\ 1011\ 0010\ 1101\ 0000\ 0110\ 0100\ 1111_2$
OP2: $4A479127_{16}$ = $0100\ 1010\ 0100\ 0111\ 1001\ 0001\ 0010\ 0111_2$
RESULT: $CAF6440B_{16}$ = $1100\ 1010\ 1111\ 0110\ 0100\ 0100\ 0000\ 1011_2$

OP1: $CB2D064F_{16}$ = $1100\ 1011\ 0010\ 1101\ 0000\ 0110\ 0100\ 1111_2$
OP2: $CA479127_{16}$ = $1100\ 1010\ 0100\ 0111\ 1001\ 0001\ 0010\ 0111_2$
RESULT: $CB5EEA98_{16}$ = $1100\ 1011\ 0101\ 1110\ 1110\ 1010\ 1001\ 1000_2$

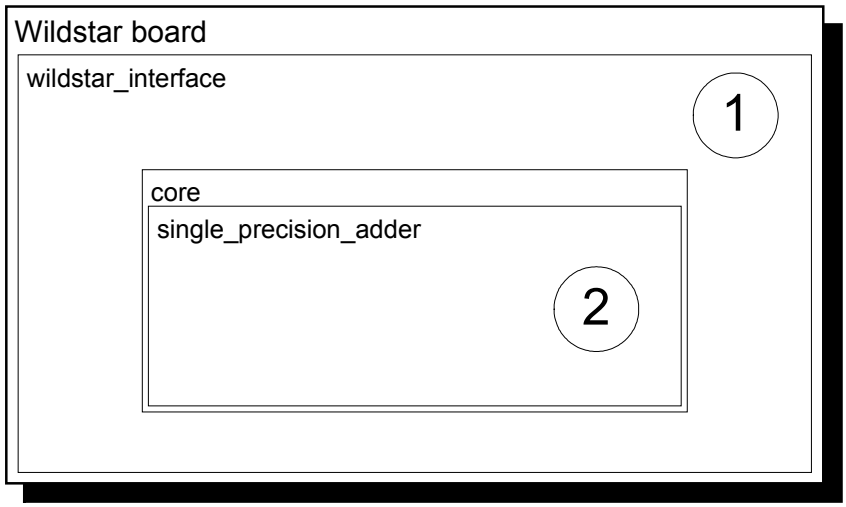
OP1: $4A312A53_{16}$ = $0100\ 1010\ 0011\ 0001\ 0010\ 1010\ 0101\ 0011_2$
OP2: $41586CE6_{16}$ = $0100\ 0001\ 0101\ 1000\ 0110\ 1100\ 1110\ 0110_2$
RESULT: $4A312A89_{16}$ = $0100\ 1010\ 0011\ 0001\ 0010\ 1010\ 1000\ 1001_2$

OP1: $4A512A53_{16}$ = $0100\ 1010\ 0101\ 0001\ 0010\ 1010\ 0101\ 0011_2$
OP2: $4A586CE6_{16}$ = $0100\ 0101\ 0101\ 1000\ 0110\ 1100\ 1110\ 0110_2$
RESULT: $4AD4CB9D_{16}$ = $0100\ 1010\ 1101\ 0100\ 1100\ 1011\ 1001\ 1101_2$

# 6.0 Synthesis Results

Module `single_precision_adder`, including all its sub-modules, has been synthesized and tested on the WildStar reconfigurable computing engine. This hardware implementation correctly operated on the same set of test vectors as presented in section 5.7.1.

The hierarchy of modules on the WildStar board begins with the `wildstar_interface` module, which contains a state machine, memory interfaces and an instance of the module core. Various architectures of the module core exist and can be chosen from. Each architecture is structural, instantiating one or more floating point modules. Of interest is the architecture `single_precision_adder_core_arch`, which instantiates exactly one `single_precision_adder` module and connects its inputs and outputs to the pipelines leading from and to memory on the WildStar board respectively. This structure is depicted below.




Hence, the only parts of this design that consume non-trivial area are the interface circuitry, indicated above as area 1, and the circuitry of the core, indicated as area 2. The former is a fixed cost, unavoidable for designs on the WildStar board and present once for how ever many functional units there are in the design. The latter is a proportional area cost, indicative of the incremental cost of adding one more functional unit to the design.

Synthesis experiments to date indicate that the fixed cost of the WildStar interface is approximately 1200 slices, while the cost of an IEEE single precision adder circuit, including denormalizing and normalizing within each module, is approximately 350 slices. The cost of a half precision adder circuit, based on 1-4-11 (s-e-f) format, is approximately 150 slices.

Given that the total area of one of the XCV 1000 processing elements on the WildStar board is 12288 slices and that a practical maximum of 85% of the chip is applicable to a normal design, due to routing overheads, the total useful area is:

$$0.85 \times 12288 \approx 10445 \text{ slices}$$

Thus, an approximate total number of `single_precision_adder` cores that can be fit onto one XCV1000 chip on the WildStar board is:

$$\frac{(10445-1200)}{350} \approx 26 \text{ functional units}$$

Similarly, an approximate total number of `half_precision_adder` cores that can be fit onto one processing element is:

$$\frac{(10445-1200)}{150} \approx 61 \text{ functional units}$$

# 7.0 Future Work

As indicated by the schedule of work in section 2.0, most of the future work in this project will be focused on synthesis of existing and new floating point arithmetic modules.

Multiply module, fp_mul, has been simulated and tested, as shown in section 5.6, and its synthesis will follow as one of the next steps in the project.

New modules for conversion to and from fixed point numbers, Fix2Float and Float2Fix, will be designed, simulated tested and synthesized.

An example application is envisaged as an important part of this project, to demonstrate the use and effectiveness of the floating point modules developed. K-means image clustering is a suitable algorithm and will be implemented using the floating point modules of this project.

# 8.0 Conclusions

Simulation, testing, synthesis and hardware implementation of floating modules in this project is progressing to the point that most of the modules have gone through the entire design process. Simulation and testing of each module has ensured correct operation of that module's description, as well as smooth transition into synthesized hardware.

Parameterized floating point addition is available in hardware, on the WildStar reconfigurable computing engine. Instances of IEEE single precision and half precision in the 1-7-11 (s-e-f) format have been synthesized. Preliminary results show that approximately 26 single precision units or 61 half precision units can fit onto a single processing element of the WildStar board.

Future work in this project includes synthesis of floating point multiplication, conversion of floating point numbers to and from fixed point, as well as an example application, K-means image clustering.

# References

IEEE, "IEEE Standard for Binary Floating-Point Arithmetic", IEEE Std 754-1985, May 21,1991

D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic", ACM Computing Surveys, Vol 23, No 1, March 1991

I. Sahin, C.S. Gloster, and C. Doss, "Feasibility of Floating-Point Arithmetic in Reconfigurable Computing Systems", 2000 MAPLD International Conference, http://rk.gsfc.nasa.gov/richcontent/MAPLDCon00/Papers/Session_E/E2_Sahin_P.pdf

Annapolis Micro Systems, Inc., "Floating-Point Math Library", Technical Data Sheet Doc # 12763-0000 Rev 1.7

I. Sahin, and C.S. Gloster, "Floating-Point Modules Targeted for Use with RC Compilation Tools", June 2001, http://www4.ncsu.edu:8030/~isahin/papers/DACPaper.pdf

N. Shirazi, A. Walters, and P. Athanas, "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines", EEE Symposium on FPGAs for Custom Computing Machines, Napa, California, Apr 1995., http://www.ccm.ece.vt.edu/papers/quantitative_text.pdf

D.A. Petterson and J.L. Hennessy, "Computer Architecture a Quantitative Approach", Second Edition, 1995

B. Fagin and C. Renard, "Field Programmable Gate Arrays and Floating Point Arithmetic", IEEE Transactions on VLSI Systems, Vol 2, No. 3, September 1994

V.P. Nelson, H.T. Nagle, B.D. Carol and J.D. Irwin, “Digital Logic Circuit Analysis and Design”, Prentice-Hall, 1995.